Part FIVE
Exceptions and Assertions

Chapter NINETEEN
Exceptions

Exam Objectives

Use try-catch and throw statements.
Use catch, multi-catch, and finally clauses.
Use Autoclose resources with a try-with-resources statement.
Create custom exceptions and Auto-closeable resources.

Exception

Errors can (and will) happen in any program. In Java, errors are represented by exceptions.

Basically, in Java there are three types of exception:

`java.lang.Exception`
Extends from `java.lang.Throwable` and represents errors that are expected. In some cases, the program can recover itself from them. Some examples are: `IOException`, `ParseException`, `SQLException`

`java.lang.RuntimeException`
Extends from `java.lang.Exception` and represents unexpected errors generated at runtime. In most cases, the program cannot recover itself from them. Some examples are: `ArithmeticException`, `ClassCastException`, `NullPointerException`

`java.lang.Error`
Extends from `java.lang.Throwable` and represents serious problems or abnormal conditions that a program should not deal with. Some examples are: `AssertionError`, `IOError`, `LinkageError`, `VirtualMachineError`

`RuntimeException` and its subclasses are not required to be caught since they're not expected all the time. They're also called unchecked.

Exception and its subclasses (except for `RuntimeException`) are known as checked exceptions because the compiler has to check if they are caught at some point by a

`try-catch` statement.

Try-Catch Block

There's only one try block

```
try {
   // Code that may throw an exception
} catch(Exception e) {
   // Do something with the exception using
   // reference e
}
```

There can be more than one catch block (one for each exception to catch)

Try-Catch Block

A `try` block is used to enclose code that might throw an exception, it doesn't matter if it's a checked or an unchecked one.

A `catch` block is used to handle an exception. It defines the type of the exception and a reference to it.

Let's see an example:

```
class Test {
    public static void main(String[] args) {
        int[] arr = new int[3];
        for(int i = 0; i <= arr.length; i++) {
            arr[i] = i * 2;
        }
        System.out.println("Done");
    }
}
```

There's an error in the above program, can you see it?

In the last iteration of the loop, `i` will be `3`, and since arrays have zero-based indexes, an exception will be thrown at runtime:

```
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 3 at com.example.T
```

If an exception is not handled, the JVM provides a default exception handler that performs the following tasks:
1. It prints out exception description.
2. It prints the stack trace (hierarchy of methods where the exception occurred).
3. It causes the program to terminate.

However, if the exception is handled by in a `try-catch` block, the normal flow of the application is maintained and rest of the code is executed.

```
class Test {
    public static void main(String[] args) {
        try {
            int[] arr = new int[3];
```

```
            for(int i = 0; i <= arr.length; i++) {
               arr[i] = i * 2;
            }
        } catch(ArrayIndexOutOfBoundsException e) {
            System.out.println("Exception caught");
        }
        System.out.println("Done");
    }
}
```

The output:

```
Exception caught
Done
```

This is an example of an unchecked exception. Again, they don't have to be caught, but catching them is certainly useful.

On the other hand, we have checked exceptions, which need to be surrounded by a `try` block if you don't want the compiler to complain. So this piece of code:

```
SimpleDateFormat sdf = new SimpleDateFormat("MM/dd");
Date date = sdf.parse("01-10"); // Compile-time error
System.out.println(date);
```

Becomes this:

```
try {
    SimpleDateFormat sdf = new SimpleDateFormat("MM/dd");
    Date date = sdf.parse("01-10");
    System.out.println(date);
} catch (ParseException e) {
    System.out.println("ParseException caught");
}
```

Since, according to its signature, the parse method throws a `java.text.ParseException` (that extends directly from `java.lang.Exception`):

```
public Date parse(String source) throws ParseException
```

The `throws` keyword indicates the exceptions that a method can throw. Only checked exceptions are required to be declared this way.

Now, remember not to confuse `throws` with `throw`. The latter will actually throw an exception:

```
public void myMethod() throws SQLException {
    // Exceptions are created with the new operator
    // like any Java class
    throw new SQLException();
}
```

We can also catch the superclass directly:

```
try {
    SimpleDateFormat sdf = new SimpleDateFormat("MM/dd");
    Date date = sdf.parse("01-10"); System.out.println(date);
} catch (Exception e) {
```

```
    System.out.println("Exception caught");
}
```

Although this is not recommended, since the above catch block will catch every exception (checked or unchecked) that could be possibly thrown by the code.

So it's better to catch both in this way:

```
try {
    SimpleDateFormat sdf = new SimpleDateFormat("MM/dd");
    Date date = sdf.parse("01-10");
    System.out.println(date);
} catch (ParseException e) {
    System.out.println("ParseException caught");
} catch (Exception e) {
    System.out.println("Exception caught");
}
```

If an exception can be caught in more than one block, the exception will be caught in the first block defined.

However, we have to respect the hierarchy of the classes, if a superclass is defined before a subclass, a compile-time error is generated:

```
try {
    SimpleDateFormat sdf = new SimpleDateFormat("MM/dd");
    Date date = sdf.parse("01-10");
    System.out.println(date);
} catch (Exception e) {
    System.out.println("Exception caught");
} catch (ParseException e) {
    System.out.println("ParseException caught");
}
```

An error is also generated if a `catch` block is defined for an exception that couldn't be thrown by the code in the `try` block:

```
try {
    SimpleDateFormat sdf = new SimpleDateFormat("MM/dd");
    Date date = sdf.parse("01-10");
    System.out.println(date);
} catch (SQLException e) { // Compile-time error
    System.out.println("ParseException caught");
}
```

The reason of these two errors is that the code of both `catch` blocks will never be executed (it's unreachable, as the compiler says).

In one case, the `catch` block with the superclass will be executed for all exceptions that belong to that type and in the other case, the exception can never be possible thrown and the `catch` block can never be possible executed.

Finally, if the code that throws a checked exception is not inside a `try-catch` block, the method that contains that code must declare the exception in the `throws` clause.

In this case, the caller of the method must either catch the exception or also declare it in the `throws` clause and so on until the main method of the program is reached:

```
public class Test {
    public static void main(String[] args)
            throws ParseException {
```

```
        m1();
    }
    private static void m1() throws ParseException {
        m2();
    }
    private static void m2() throws ParseException {
        m3();
    }
    private static void m3() throws ParseException {
        m4();
    }
    private static void m4() throws ParseException {
        SimpleDateFormat sdf =
             new SimpleDateFormat("MM/dd");
        Date date = sdf.parse("01-10");
        System.out.println(date);
    }
}
```

Multi-Catch

```
try {
    // Code that may throw one or
    // two exceptions
} catch(Exception1 | Exception2 e) {
    // Do something with the caught
    // exception using reference e
}
```

Catch either Exception1 or Exception2

Finally

```
try {
    // Code that may throw an
    // exception }
finally {
    // Block that is always executed
}
```

The catch block is optional. You can have both or either a catch block or a finally block

The finally block is always executed, no matter if an exception is thrown in the try block, re-thrown inside the catch block, or not caught at all

---

Multi-Catch and Finally

Consider something like the following code:

```
int res = 0;
try {
    int[] arr = new int[2];
    res = (arr[1] != 0) ? 10 / arr[1] : 10 * arr[2];
} catch (ArithmeticException e) {
    e.printStackTrace();
    return res;
} catch (IndexOutOfBoundsException e) {
    e.printStackTrace();
    return res;
```

```
}
return res;
```

Isn't that ugly? I mean to have two catch blocks with the same code. Fortunately, the `multi-catch` block allows us to catch two or more exception with a single `catch` block:

```
try {
    ...
} catch (ArithmeticException | IndexOutOfBoundsException e) {
    e.printStackTrace();
    return res;
}
```

Think of the pipe character as an **OR** operator. Also, notice there's only one variable at the end of the `catch` clause for all the exceptions declared. If you want to differentiate between exceptions, you can use the `instanceof` operator:

```
try {
   ...
} catch (ArithmeticException | IndexOutOfBoundsException e) {
    if(e instanceof ArithmeticException) {
        // Do something else if the exception type
        // is ArithmeticException
    }
    e.printStackTrace();
    return res;
}
```

Also, the variable is treated as final, which means that you can't reassign (why would you want anyway?):

```
try {
    ...
} catch (ArithmeticException | IndexOutOfBoundsException e) {
    if(e instanceof ArithmeticException) {
        // Compile-time error
        e = new ArithmethicException("My Exception");
    }
} catch(Exception e) {
    e = new Exception("My Exception"); // It compiles!
    throw e;
}
```

One last rule. You cannot combine subclasses and their superclasses in the same `multi-catch` block:

```
try {
    ...
} catch (ArithmeticException | RuntimeException e) {
    // The above line generates a compile-time error
    // because ArithmeticException is a subclass of
    // RuntimeException
    e.printStackTrace();
    return res;
}
```

This is similar to the case when a superclass is declared in a `catch` block before the subclass. The code is redundant, the superclass will always catch the exception.

Back to this piece of code:

```
int res = 0;
try {
    int[] arr = new int[2];
    res = (arr[1] != 0) ? 10 / arr[1] : 10 * arr[2];
} catch (ArithmeticException | IndexOutOfBoundsException e) {
    e.printStackTrace();
    return res;
}
return res;
```

Since the value of `res` is always returned, we can use a `finally` block:

```
try {
    ...
} catch (ArithmeticException | IndexOutOfBoundsException e) {
    e.printStackTrace();
} finally {
    return res;
}
```

The `finally` block is **ALWAYS** executed, even when an exception is caught or when either the `try` or `catch` block contains a return statement. For that reason, it's commonly used to close resources like database connections or file handlers.

There's only one exception to this rule. If you call `System.exit()` , the program will terminate abnormally without executing the `finally` block. However, as it's considered bad practice to call `System.exit()` , this rarely happens.

Try-With-Resources

Resource that is closed automatically

```
try (AutoCloseableResource r = new AutoCloseableResource()) {
    // Code that may thrown an exception
} catch(Exception e) {
    // Handle exception
} finally {
    // Always executes
}
```

Resource is closed after the try block finishes

Catch and finally blocks are both optional in a try-with-resources

---

try-with-resources

As we said before, the `finally` block is generally used to close resources. Since Java 7, we have the `try-with-resources` block, in which the `try` block, one or more resources are declared so they can be closed without doing it explicitly in a `finally` block:

```
try (BufferedReader br =
       new BufferedReader(new FileReader("/file.txt"))) {
    int value = 0;
    while((value = br.read()) != -1) {
        System.out.println((char)value);
    }
} catch (IOException e) {
```

```
    e.printStackTrace();
}
```

In the example, the `BufferedReader` is closed after the `try` block finishes its execution. This would be equivalent to:

```
BufferedReader br = null;
try {
    int value = 0;
    br = new BufferedReader(new FileReader("/file.txt"));
    while((value = br.read()) != -1) {
        System.out.println((char)value);
    }
} catch (IOException e) {
    e.printStackTrace();
} finally {
    try {
        if (br != null) br.close();
    } catch (IOException e) {
        e.printStackTrace();
    }
}
```

If you declare more than one resource, they have to be separated by a semicolon:

```
try (FileReader fr = new FileReader("/file.txt");
        BufferedReader br = new BufferedReader(fr)) {
    ...
}
```

Also, resources declared inside a `try-with-resources` cannot be used outside this block (first reason, they're out of scope, second reason, they're closed after the `try` block ends):

```
try (BufferedReader br =
        new BufferedReader(new FileReader("/file.txt"))) {
    ...
}
String line = br.readLine(); // Compile-time error
```

Now, don't think any class will work in a `try-with-resources` .

```
class MyResource {
    void useResource() { }
}
...
try (MyResource r = new MyResource()) { // Compile-time error
    r.useResource()
}
```

The class(es) used in a `try-with-resources` block must implement one of the following interfaces:

- `java.lang.AutoClosable`
- `java.io.Closable`

They both declare a `close()` method, and the only practical difference between these two interfaces is that the close method of the `Closeable` interface only throws exceptions of type `IOException` :

```
void close() throws IOException;
```

While the `close()` method of the `AutoCloseable` interface throws exceptions of type `Exception` (in other words, it can throw almost any kind of exception):

```
void close() throws Exception;
```

So the `close()` method is called automatically, and if this method actually throws and exception, we can catch it in the `catch` block.

```
class MyResource implements AutoCloseable {
    public void close() throws Exception {
        int x = 0;
        //...
        if(x == 1) throw new Exception("Close Exception");
    }
    void useResource() {}
}
...
try (Resource r = new Resource()) { // Problem gone!
    r.useResource();
} catch (Exception e) {
    e.printStackTrace();
}
```

But what happens if the `try` block also throws an exception?

Well, the result is that the exception from the `try` block "wins" and the exceptions from the `close()` method are "suppressed".

In fact, you can retrieve these suppressed exceptions by calling the `Throwable[] java.lang.Throwable.getSuppressed()` method from the exception thrown by the `try` block.

```
try (Resource r = new Resource()) {
    r.useResource();
    throw new Exception("Exception inside try");
} catch (Exception e) {
    System.out.println(e.getMessage());
    Stream.of(e.getSuppressed())
        .forEach(t -> System.out.println(t.getMessage()));
}
```

The output (assuming the `close()` method throws an exception):

```
Exception inside try
Close Exception
```

Custom exceptions

Since exceptions are classes, we can just extend any exception of the language to create our own exceptions.

If you want to force the catching of your exception, extend from `Exception` or one of its subclasses. If you don't want to force it, extend from `RuntimeException` or one of its subclasses.

```
class TooHardException extends Exception {
    public TooHardException(Exception e) {
        super(e);
    }
}
class TooEasyException extends RuntimeException { }
```

As you can see, it's a convention to add `Exception` to your classes' name. The `Error` and `Throwable` classes are not actually used for custom exceptions.

The main members of the `Exception` class that you'd want to know are:

Description	Constructor/Method
Default constructor	`Exception()`
Constructor that takes a message	`Exception(String)`
Constructor that takes another exception (that represents the cause)	`Exception(Throwable)`
Returns exception's message	`String getMessage()`
Returns (if any) the exception's cause	`Throwable getCause()`
Returns the list of suppressed exceptions	`Throwable[] getSuppressed()`
Prints the stack trace (cause and suppressed exceptions included)	`void printStackTrace()`

Key Points

- In Java, there are three types of exception
  - `java.lang.Exception`
  - `java.lang.RuntimeException`
  - `java.lang.Error`
- `RuntimeException` and its subclasses are not required to be caught since they're not expected all the time. They're also called unchecked.
- `Exception` and its subclasses (except for `RuntimeException`) are known as checked exceptions because the compiler has to check if they are caught at some point by a `try-catch` statement.
- If an exception can be caught in more than one block, the exception will be caught in the first block defined.
- However, we have to respect the hierarchy of the classes, if a superclass is defined before a subclass, a compile-time error is generated.
- If the code that throws a checked exception is not inside a `try-catch` block, the method that contains that code must declare the exception in the `throws` clause.
- In this case, the caller of the method must either catch the exception or also declare it in the `throws` clause and so on until the main method of the program is reached.
- The `multi-catch` block allows us to catch two or more unrelated exceptions with a single `catch` block:

  ```
  try {
      // ...
  } catch(Exception1 | Exception2 e) {
      // ...
  }
  ```

- The finally block is **ALWAYS** executed, even when an exception is caught or when either the try or catch block contains a `return` statement.
- In a `try-with-resources` block, one or more resources are declared so they can be closed automatically after the `try` block ends just by implementing

`java.lang.AutoCloseable` or `java.io.Closeable`:

```
try (Resource r = new Resource()) {
    //...
} catch(Exception e) { }
```

- When using a `try-with-resources` block, `catch` and `finally` are optional. You can create your own exceptions just by extending from `java.lang.Exception` (for checked exceptions) or `java.lang.RuntimeException` (for unchecked exceptions).

Self Test

1. Given:

```
public class Question_19_1 {
    protected static int m1() {
        try {
            throw new RuntimeException();
        } catch(RuntimeException e) {
             return 1;
        } finally {
             return 2;
        }
    }
    public static void main(String[] args) {
        System.out.println(m1());
    }
}
```

What is the result?
A. `1`
B. `2`
C. Compilation fails
D. An exception occurs at runtime

2. Given:

```
public class Question_19_2 {
    public static void main(String[] args) {
        try {
            // Do nothing
        } finally {
            // Do nothing
        }
    }
}
```

What of the following is true?
A. The code doesn't compile correctly
B. The code would compile correctly if we add a `catch` block
C. The code would compile correctly if we remove the `finally` block
D. The code compiles correctly as it is

3. Which of the following statements are true?
A. In a `try-with-resources`, the `catch` block is required.
B. The `throw`s keyword is used to throw an exception.
C. In a `try-with-resources` block, if you declare more than one resource, they have to be separated by a semicolon.
D. If a `catch` block is defined for an exception that couldn't be thrown by the code in the `try` block, a compile-time error is generated.

4. Given:

```
class Connection implements java.io.Closeable {
    public void close() throws IOException {
        throw new IOException("Close Exception");
    }
}

public class Question_19_4 {
    public static void main(String[] args) {
        try (Connection c = new Connection()) {
            throw new RuntimeException("RuntimeException");
        } catch (IOException e) {
            System.out.println(e.getMessage());
        }
    }
}
```

What is the result?
A. Close `Exception`
B. `RuntimeException`
C. `RuntimeException` and then `CloseException`
D. Compilation fails
E. The stack trace of an uncaught exception is printed

5. Which of the following exceptions are direct subclasses of `RuntimeException` ?
A. `java.io.FileNotFoundException`
B. `java.lang.ArithmeticException`
C. `java.lang.ClassCastException`
D. `java.lang.InterruptedException`

Open answers page

---

18. Parallel Streams 20. Assertions